Introduction to

PYTHON PROGRAMING

TUPLES, LISTS, AND DICTIONARIES

A **DATA STRUCTURE** MODELS A COLLECTION OF DATA, SUCH AS A LIST OF NUM BERS, A ROW IN A SPREADSHEET, OR A RECORD IN A DATABASE.

TUPLES ARE IMMUTABLESEQUENCES

A **SEQUENCE** IS AN ORDERED LIST OF VALUES. EACH ELEMENT IN A SEQUENCE IS ASSIGNED AN INTEGER, CALLED AN **INDEX,** THAT DETERMINES THE ORDER IN WHICH THE VALUES APPEAR

WHAT IS A TUPLE?

THE WORD **TUPLE** COMES FROM MATHEMATICS,WHERE IT IS USED TO DESCRIBE A FINITE ORDERED SEQUENCE OF VALUES.

HOW TO CREATE A TUPLE

1. TUPLE LITERALS
2. THE TUPLE () BUILT-IN

TUPLE LITERALS

A TUPLE LITERAL IS A TUPLE THAT IS WRITTEN OUT EXPLICITLY ASACOMMA-SEPARATED LIST OF VALUES SURROUNDED BY PARENTHESES.

```
>>> my_first_tuple = (1, 2, 3)
```

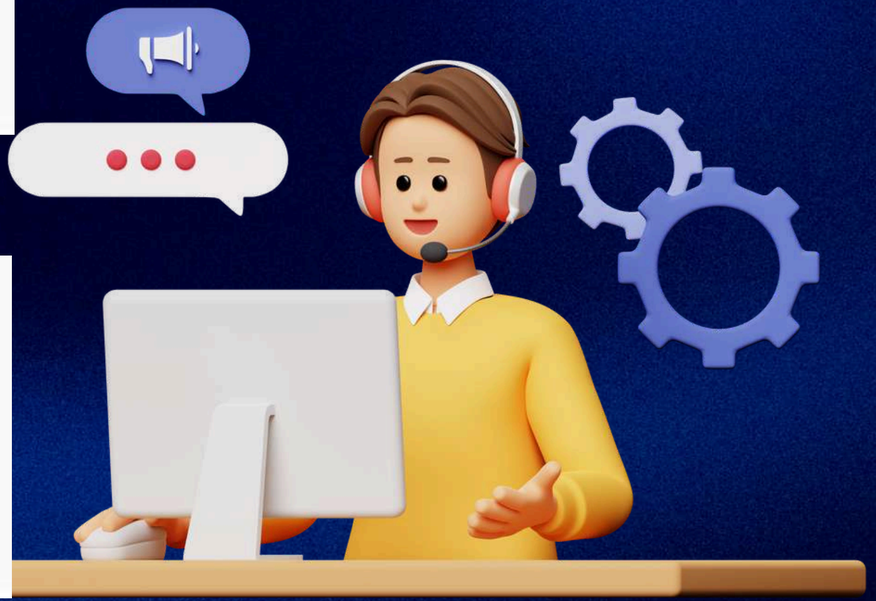
THERE IS A SPECIAL TUPLE THAT DOESN'T CONTAIN ANY VALUES. THIS TUPLE IS CALLED THE **EMPTY TUPLE** AND CAN BE CREATED BY TYPING TWO PARENTHESES WITHOUT ANYTHING BETWEEN THEM:

```
>>> empty_tuple = ()
```

THE TUPLE () BUILT-IN

```
>>> tuple("Python")
('P', 'y', 't', 'h', 'o', 'n')
```

```
>>> tuple(1, 2, 3)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    tuple(1, 2, 3)
TypeError: tuple expected at most 1 arguments, got 3
```



```
>>> tuple(1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    tuple(1)
TypeError: 'int' object is not iterable
```

THE WORD **ITERABLE** IN THE ERROR MESSAGE INDICATES THAT A SINGLE INTEGER CAN'T BE **ITERATED**, WHICH IS TO SAY THAT THE INTEGER DATA TYPE DOESN'T CONTAIN MULTIPLE VALUES THAT CAN BE ACCESSED ONE-BY-ONE

SIMILARITIES BETWEEN TUPLES AND STRINGS

- TUPLES HAVE ALENGTH

```
>>> numbers = (1, 2, 3)
>>> len(numbers)
3
```

- **TUPLES SUPPORT INDEXING AND SLICING**

```
>>> name = "David"
>>> name[1]
'a'
```

```
>>> name = "David"
>>> name[2:4]
"vi"
```

TUPLES ARE IMMUTABLE

THIS MEANS YOU CAN'T CHANGE THE VALUE OF AN ELEMENT OF A TUPLE ONCE IT HAS BEEN CREATED.

```
>>> values[0] = 2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    values[0] = 2
TypeError: 'tuple' object does not support item assignment
```

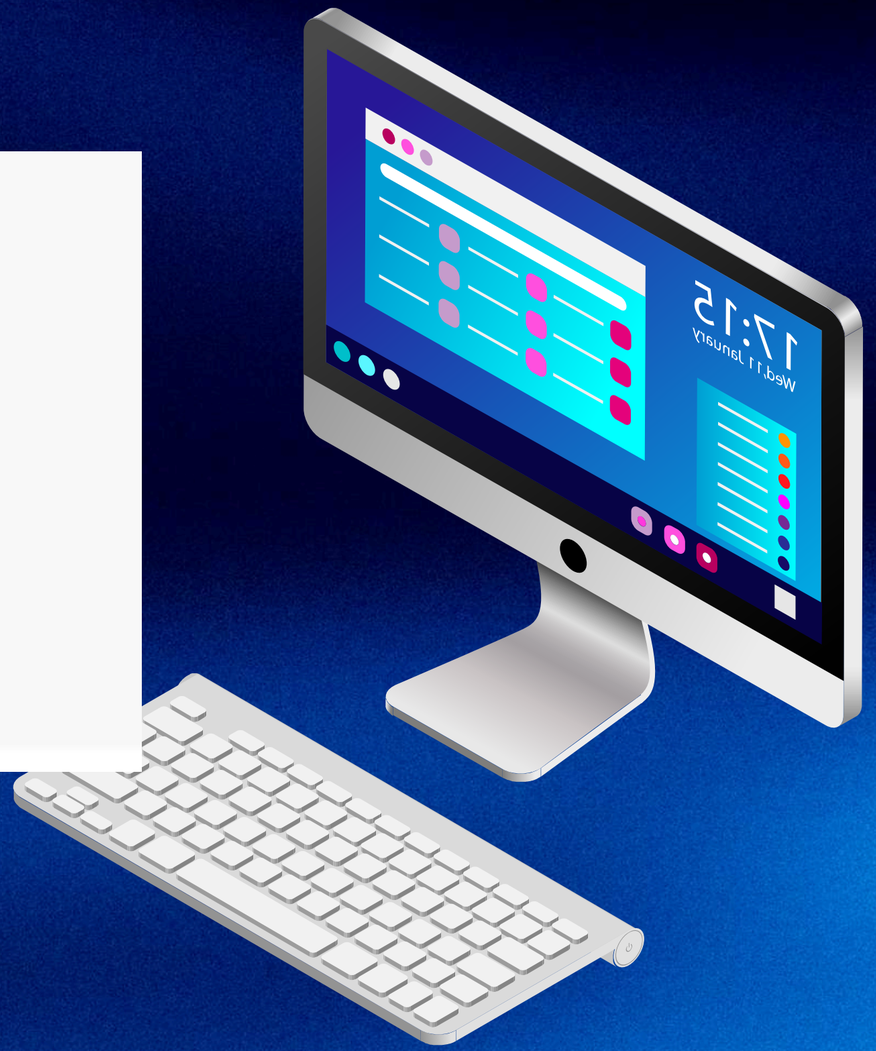
Note

Although tuples are immutable, there are some situations in which the values in a tuple can change.

These quirks and oddities are covered in depth in Real Python's Immutability in Python video course.

TUPLES ARE ITERABLE

```
>>> vowels = ("a", "e", "i", "o", "u")
>>> for vowel in vowels:
...     print(vowel.upper())
...
A
E
I
O
U
```



TUPLE PACKING AND UNPACKING

```
>>> coordinates = 4.21, 9.29
>>> type(coordinates)
<class 'tuple'>
```

IT LOOKS LIKE TWO VALUES ARE BEING ASSIGNED TO THE SINGLE VARIABLE *COORDINATES*. IN A SENSE, THEY ARE, ALTHOUGH THE RESULT IS THAT BOTH VALUES ARE **PACKED** INTO A SINGLE TUPLE.

```
>>> x, y = coordinates
>>> x
4.21
>>> y
9.29
```

HERE THE VALUES CONTAINED IN THE SINGLE TUPLE COORDINATES ARE **UN PACKED** INTO TWO DISTINCT VARIABLES X AND Y.

Note

While assigning multiple variables in a single line can shorten the number of lines in a program, you may want to refrain from assigning too many values in a single line.

Assigning more than two or three variables this way can make it difficult to tell which value is assigned to which variable name.

CHECKING EXISTENCE OF VALUES WITH *IN*

```
>>> vowels = ("a", "e", "i", "o", "u")
>>> "o" in vowels
True
>>> "x" in vowels
False
```

IF THE VALUE TO THE LEFT OF IN IS CONTAINED IN THE TUPLE TO THE RIGHT OF IN, THE RESULT IS *TRUE*. OTHERWISE, THE RESULT IS *FALSE*.

RETURNING MULTIPLE VALUES FROM A FUNCTION

```
>>> def adder_subtractor(num1, num2):
...     return (num1 + num2, num1 - num2)
...
>>> adder_subtractor(3, 2)
(5, 1)
```

THEFUNCTION *ADDER_SUBTRACTOR ()* HAS TWO PARAMETERS,*NUM1* AND *NUM2*,AND RETURNS A TUPLE WHOSE FIRST ELEMENT IS THE SUM OF THE TWO NUMBERS,AND WHOSE SECOND ELEMENT IS THE DIFFERENCE.

REVIEW EXERCISES

1. CREATE ATUPLE LITERAL NAMED CARDINAL_NUMBERS THAT HOLDS THE STRINGS "FIRST", "SECOND" AND "THIRD", IN THAT ORDER.

2. USING INDEX NOTATION AND PRINT(), DISPLAY THE STRING AT INDEX 1 IN CARDINAL_NUMBERS.

3. UNPACK THE VALUES IN CARDINAL_NUMBERS INTO THREE NEW STRINGS NAMED POSITION1, POSITION2 AND POSITION3 IN A SINGLE LINE OF CODE, THEN PRINT EACH VALUE ON A SEPARATE LINE.

4. CREATE ATUPLE CALLED MY_NAME THAT CONTAINS THE LETTERS OF YOUR NAME BY USING TUPLE() AND A STRING LITERAL.

5. CHECK WHETHER OR NOT THE CHARACTER "X" IS IN MY_NAME USING THE IN KEYWORD.

6. CREATEANEWTUPLECONTAININGALLBUTTHEFIRSTLETTERINMY_NAMEUSING SLICING NOTATION.

LISTS ARE MUTABLE SEQUENCES

THE *LIST* DATA STRUCTURE IS ANOTHER SEQUENCE TYPE IN PYTHON. JUST LIKE STRINGS AND TUPLES, LISTS CONTAIN ITEMS THAT ARE INDEXED BY INTEGERS, STARTING WITH 0.

LISTS ARE **MUTABLE**, MEANING YOU CAN CHANGE

CREATING LISTS

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> type(colors)
<class 'list'>
```

A **LIST LITERAL** LOOKS ALMOST EXACTLY LIKE A TUPLE LITERAL, EXCEPT THAT IT IS SURROUNDED WITH SQUARE BRACKETS ([AND]) INSTEAD OF PARENTHESES

THREE WAYS TO CREATE A LIST:

1. A LIST LITERAL
2. THE LIST() BUILT-IN
3. THE STRING .SPLIT() METHOD

BASIC LIST OPERATIONS

YOU CAN ACCESS LIST ELEMENTS USING INDEX NOTATION:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1]
2
```

- YOU CAN CREATE A NEW LIST FROM AN EXISTING ONCE USING SLICE NOTATION:

```
>>> # Check existence of an element
>>> "Bob" in numbers
False
```

- BECAUSE LISTS ARE ITERABLE, YOU CAN ITERATE OVER THEM WITH A FOR LOOP.

```
>>> # Print only the even numbers in the list
>>> for number in numbers:
...     if number % 2 == 0:
```

```
...         print(number)
...
2
4
```

CHANGING ELEMENTS IN A LIST

THE ABILITY TO SWAP VALUES IN A LIST FOR OTHER VALUES IS CALLED **MUTABIL ITY.** LISTS ARE **MUTABLE**. THE ELEMENTS OF TUPLES MAY NOT BE SWAPPED FOR NEW VALUES, SO TUPLES ARE SAID TO BE **IMMUTABLE**.

TO SWAP A VALUE IN A LIST WITH ANOTHER, ASSIGN THE NEW VALUE TO A SLOT USING INDEX NOTATION:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[0] = "burgundy"
```

YOU CAN CHANGE SEVERAL VALUES IN A LIST AT ONCE WITH A SLICE ASSIGNMENT:

```
>>> colors[1:3] = ["orange", "magenta"]
>>> colors
['burgundy', 'orange', 'magenta', 'blue']
```

LIST METHODS FOR ADDING AND REMOVING ELEMENTS

- LIST.INSERT()

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> # Insert "orange" into the second position
>>> colors.insert(1, "orange")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue']
```

Important

When you `.insert()` an item into a list, you do not need to assign the result to the original list.

For example, the following code actually erases the colors list:

```
>>> colors = colors.insert(-1, "indigo")
>>> print(colors)
None
```

`.insert()` is said to alter `colors` **in place**. This is true for all list methods that do not return a value.

- LIST.POP()

```
>>> color = colors.pop(3)
>>> color
'green'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

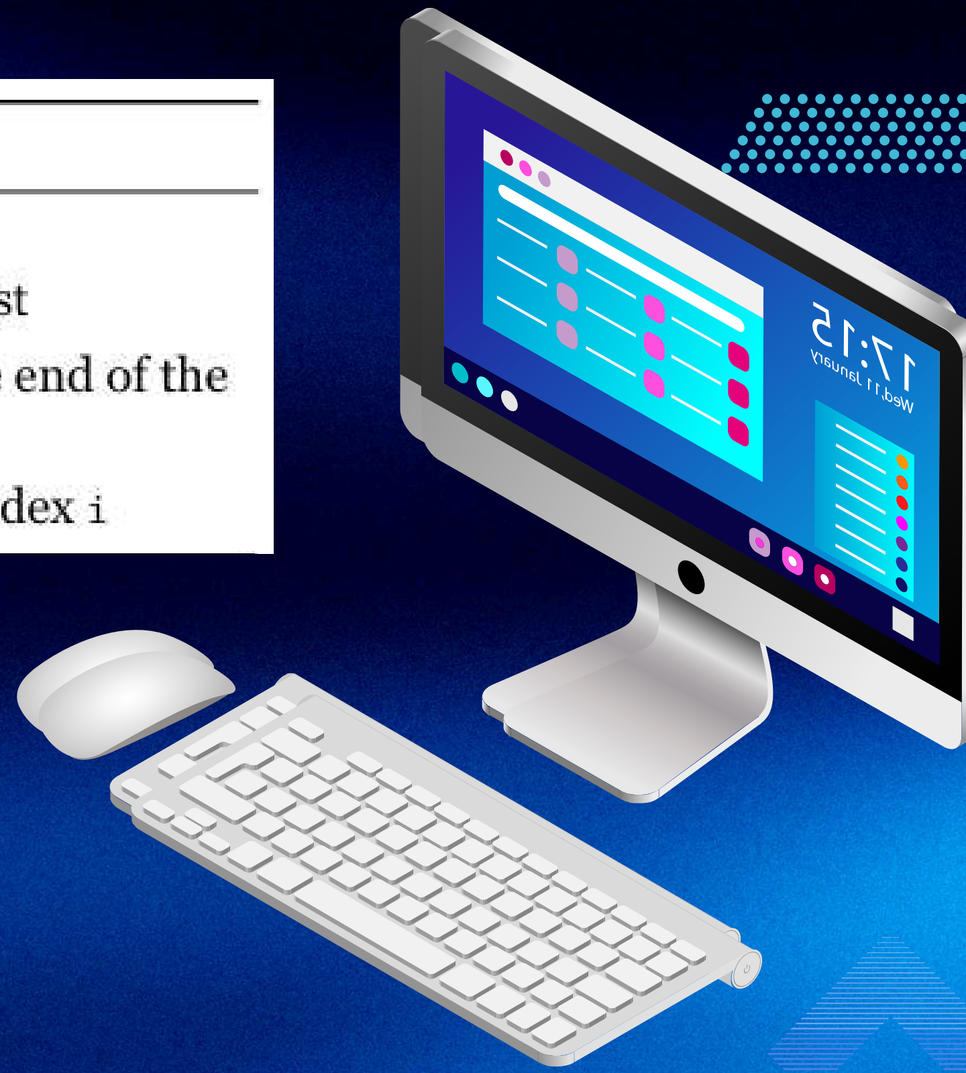
- **LIST.APPEND()**

```
>>> colors.append("indigo")
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

- **LIST.EXTEND()**

```
>>> colors.extend(["violet", "ultraviolet"])
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet', 'ultraviolet']
```

List Method	Description
`.insert(i, x)`	Insert the value `x` at index `i`
`.append(x)`	Insert the value `x` at the end of the list
`.extend(iterable)`	Insert all the values of `iterable` at the end of the list, in order
`.pop(i)`	Remove and return the element at index `i`



LISTS OF NUMBERS

YOU CAN DOTHIS WITH A FOR LOOP:

```
>>> nums = [1, 2, 3, 4, 5]
>>> total = 0
>>> for number in nums:
...     total = total + number
...
>>> total
15
```

ALTHOUGH THIS FOR LOOP IS STRAIGHTFORWARD, THERE IS A MUCH MORE SUC CINCT WAY OF DOING THIS IN PYTHON:

```
>>> sum([1, 2, 3, 4, 5])
15
```

BESIDES SUM(),THERE ARE TWO OTHER USEFUL BUILT-IN FUNCTIONS FOR WORKING WITH LISTS OF NUMBERS: *MIN() AND MAX()*.THESEFUNCTIONSRETURNTHEMINIMUMANDMAXIMUMVALUESINTHELIST,RESPECTIVELY:

```
>>> min([1, 2, 3, 4, 5 ])
1

>>> max([1, 2, 3, 4, 5])
5
```

```
>>> sum((1, 2, 3, 4, 5))
15

>>> min((1, 2, 3, 4, 5))
1

>>> max((1, 2, 3, 4, 5))
5
```

LIST COMPREHENSIONS

YET ANOTHER WAY TO CREATE A LIST FROM AN EXISTING ITERABLE IS WITH A **LIST COMPREHENSION:**

```
>>> numbers = (1, 2, 3, 4, 5)
>>> squares = [num**2 for num in numbers]
```

```
>>> squares
[1, 4, 9, 16, 25]
```

TO CREATE THE *SQAURES* LIST USINGA TRADITIONAL FOR LOOP INVOLVES FIRST CREATING ANEMPTYLIST, LOOPING OVER THE NUMBERS IN NUMBERS, AND APPENDING THE SQUARE OF EACH NUMBER TO THE LIST:

```
>>> squares = []
>>> for num in numbers:
...     sqaures.append(num**2)
...
>>> squares
[1, 4, 9, 16, 25]
```

FOR INSTANCE, SUPPOSE YOU NEEDED TO CONVERT A LIST OF STRINGS CONTAIN ING FLOATING POINT VALUES TO A LIST OF FLOAT OBJECTS. THE FOLLOWING LIST COMPREHENSIONS ACHIEVES THIS:

```
>>> str_numbers = ["1.5", "2.3", "5.25"]
>>> float_numbers = [float(value) for value in str_numbers]
>>> float_numbers
[1.5, 2.3, 5.25]
```

REVIEW EXERCISES

1. CREATE A LIST NAMED FOOD WITH TWO ELEMENTS "RICE" AND "BEANS".

2. APPENDTHE STRING "BROCCOLI" TO FOOD USING .APPEND().

3. ADDTHESTRING "BREAD" AND "PIZZA" TO "FOOD" USING .EXTEND().

4. PRINT THE FIRST TWO ITEMS IN THE FOOD LIST USING PRINT() AND SLICING NOTATION.

5. PRINT THE LAST ITEM IN FOOD USING PRINT() AND INDEX NOTATION.

6. CREATE A LIST CALLED BREAKFAST FROM THE STRING "EGGS, FRUIT, ORANGE JUICE" USING THE STRING .SPLIT() METHOD.

7. VERIFY THAT BREAKFAST HAS THREE ITEMS USING LEN().

8. CREATE A NEW LIST CALLED LENGTHS USING A LIST COMPREHENSION THAT CONTAINS THE LENGTHS OF EACH STRING IN THE BREAKFAST LIST.

NESTING, COPYING, AND SORTING TUPLES AND LISTS

- ***NESTING LISTS AND TUPLES***

A **NESTED LIST**,OR **NESTED**UPLE, IS A LIST OR TUPLE THAT IS CONTAINED AS A VALUE IN ANOTHER LIST OR TUPLE.

```
>>> two_by_two = [[1, 2], [3, 4]]

>>> # two_by_two has length 2
>>> len(two_by_two)
2

>>> # Both elements of two_by_two are lists
>>> two_by_two[0]
[1, 2]
>>> two_by_two[1]
[3, 4]
```

Note

Readers interested in data analysis or scientific computing may recognize lists of lists as a sort of matrix of values.

While you can use the built in `list` and `tuple` types for matrices, better alternatives exist. To learn how to work with matrices in Python, check out Chapter 17.

- ***COPYING A LIST***

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals
>>> large_cats.append("Tigger")
>>> animals
['lion', 'tiger', 'frumious Bandersnatch', 'Tigger']
```

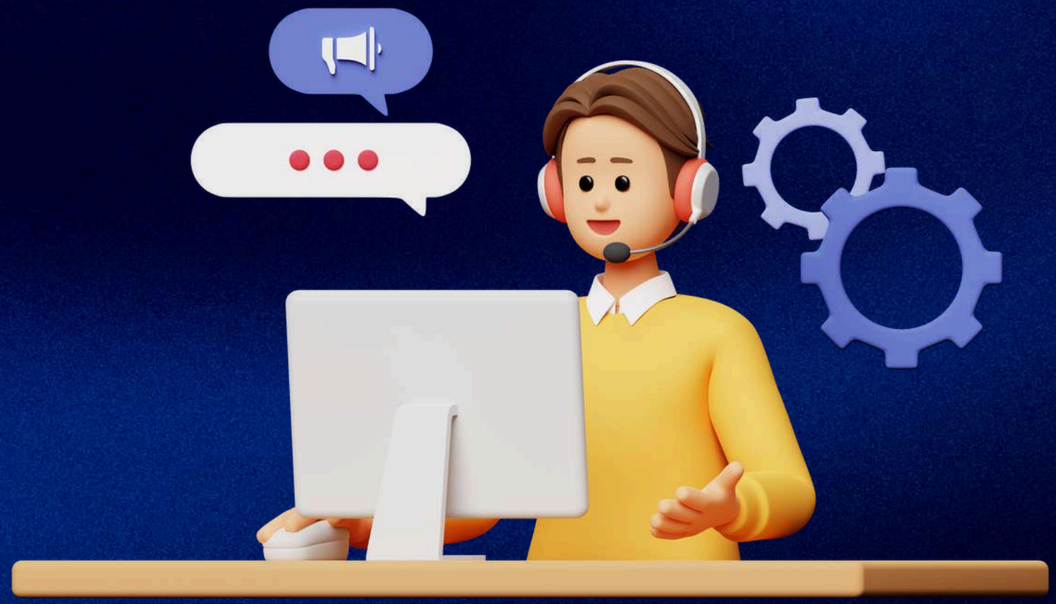
IN PROGRAMMING JARGON, THIS METHOD OF COPYING A LIST IS CALLED A **SHALLOW COPY.**

TO MAKE A COPY OF BOTH THE LIST AND ALL OF THE ELEMENTS IT CONTAINS, YOU MUST USE WHAT IS KNOWN AS A **DEEP COPY.**

- ***SORTING LISTS***

```
>>> # Lists of strings are sorted alphabetically
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors.sort()
>>> colors
['blue', 'green', 'red', 'yellow']

>>> # Lists of numbers are sorted numerically
>>> numbers = [1, 10, 5, 3]
>>> numbers.sort()
>>> numbers
[1, 3, 5, 10]
```



REVIEW EXERCISES

1. CREATE A TUPLE DATA WITH TWO VALUES. THE FIRST VALUE SHOULD BE THE TUPLE (1, 2) AND THE SECOND VALUE SHOULD BE THE TUPLE (3, 4).

2. WRITE A FOR LOOP THAT LOOPS OVER DATA AND PRINTS THE SUM OF EACH NESTED TUPLE. THE OUTPUT SHOULD LOOK LIKE THIS:

```
Row 1 sum: 3
Row 2 sum: 7
```

4. CREATE A COPY OF THE NUMBERS LIST USING THE [:] SLICING NOTATION.

5. SORT THE NUMBERS LIST IN NUMERICAL ORDER USING THE .SORT() METHOD

CHALLENGE: LIST OF LISTS

WRITE A PROGRAM THAT CONTAINS THE FOLLOWING LISTS OF LISTS:

```
universities = [
    ['California Institute of Technology', 2175, 37704],
    ['Harvard', 19627, 39849],
    ['Massachusetts Institute of Technology', 10566, 40732],
    ['Princeton', 7802, 37000],
    ['Rice', 5879, 35551],
    ['Stanford', 19535, 40569],
    ['Yale', 11701, 40500]
]
```

CHALLENGE: WAX POETIC

CREATE FIVE LISTS FOR DIFFERENT WORD TYPES:

· NOUNS: ["FOSSIL", "HORSE", "AARDVARK", "JUDGE", "CHEF", "MANGO","EXTROVERT", "GORILLA"]

· VERBS: ["KICKS", "JINGLES", "BOUNCES", "SLURPS", "MEOWS", "EXPLODES", "CURDLES"]

· ADJECTIVES: ["FURRY", "BALDING", "INCREDULOUS", "FRAGRANT", "EXUBERANT", "GLISTENING"]

· PREPOSITIONS: ["AGAINST", "AFTER", "INTO", "BENEATH", "UPON", "FOR", "IN", "LIKE", "OVER", "WITHIN"]

· ADVERBS: ["CURIOUSLY", "FURIOUSLY", "SENSUOUSLY"] "EXTRAVAGANTLY", "TANTALIZINGLY"

Store Relationships in Dictionaries

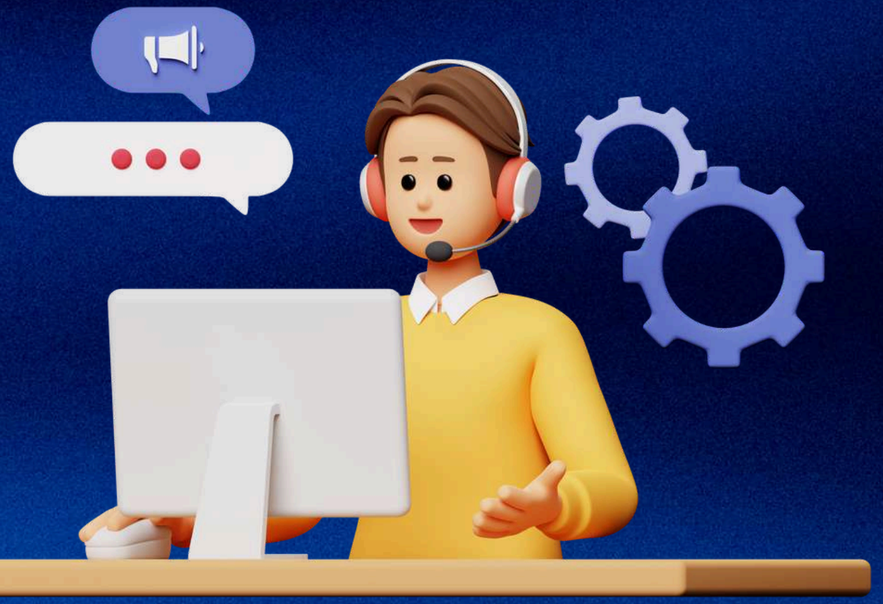
WHAT IS A DICTIONARY?

DICTIONARIES HOLD INFORMATION IN PAIRS OF DATA CALLED **KEY-VALUE PAIRS**. THAT IS, EACH OBJECT IN A DICTIONARY HAS TWO PARTS: A **KEY** AND A **VALUE**

THE **KEY** IN A KEY-VALUE PAIR IS A UNIQUE NAMET HAT IDENTIFIES THE **VALUE** PART OF THE PAIR.

Key	Value
"California"	"Sacramento"
"New York"	"Albany"

Key	Value
"Texas"	"Austin"



A PYTHON DICTIONARY IS THAT THE RELATIONSHIP BETWEEN A KEY AND ITS VALUE IS COMPLETELY ARBI TRARY.

Key	Value
1	"Sunday"
"red"	12:45pm
17	True

CREATING DICTIONARIES

```
>>> capitals = {
  "California": "Sacramento",
  "New York": "Albany",
  "Texas": "Austin",
}
```

```
>>> key_value_pairs = (
...     ("California", "Sacramento"),
...     ("New York", "Albany"),
...     ("Texas", "Austin"),
)
>>> capitals = dict(key_value_pairs)
```

Note

If you happen to be following along with a Python version older than 3.6, then you will notice that the output dictionaries in the interactive window have a different order than the ones that appear in these examples.

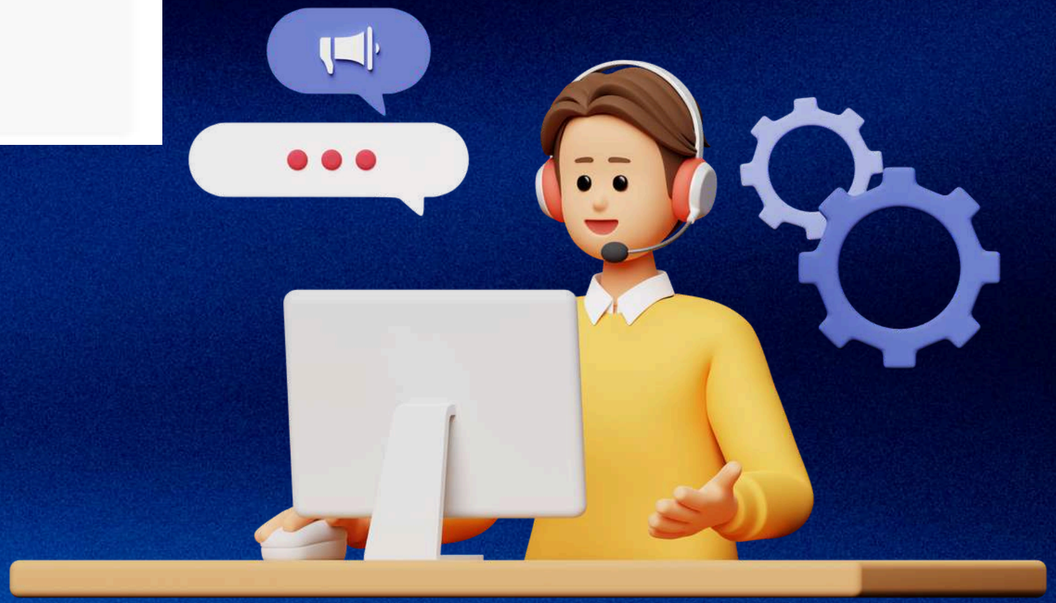
Prior to Python 3.6, the order of key-value pairs in a Python dictionary was random. In later versions, the order of the key-value pairs is guaranteed to match the order in which they were inserted.

ACCESSING DICTIONARY VALUES

```
>>> capitals["Texas"]
'Austin'
```

```
>>> capitals_list[0]  # Capital of California
'Sacramento'

>>> capitals_list[2]  # Capital of Texas
'Austin'
```



ADDING AND REMOVING VALUES IN A DICTIONARY

LET'S ADD THE CAPITAL OF COLORADO TO THE CAPITALS DICTIONARY:

```
>>> capitals["Colorado"] = "Denver"
```

```
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Austin',
'Colorado': 'Denver'}
```

```
>>> capitals["Texas"] = "Houston"
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Houston',
'Colorado': 'Denver'}
```

TO REMOVE AN ITEM FROM A DICTIONARY,USE THE DEL KEYWORD WITH THE KEY FOR THE VALUE YOU WANT TO DELETE:

```
>>> del capitals["Texas"]
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany',
'Colorado': 'Denver'}
```

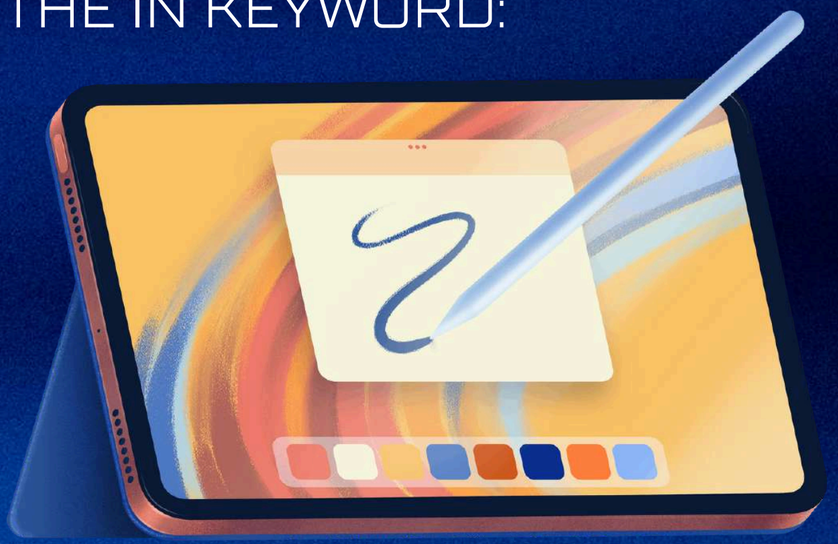
CHECKING THE EXISTENCE OF DICTIONARY KEYS

```
>>> capitals["Arizona"]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    capitals["Arizona"]
KeyError: 'Arizona'
```

THE *KEY ERROR* IS THE MOST COMMON ERROR ENCOUNTERED WHEN WORKING WITH DICTIONARIES

YOU CAN CHECK THAT A KEY *EXISTS IN* A DICTIONARY USING THE IN KEYWORD:

```
>>> "Arizona" in capitals
False
>>> "California" in capitals
True
```



ITERATING OVER DICTIONARIES

WHEN YOU LOOP OVER A DICTIONARY WITH A *FOR* LOOP,YOU ITERATE OVER THE DICTIONARY'S KEYS:

```
>>> for key in capitals:
...     print(key)
...
California
New York
Colorado
```

```
>>> for state in capitals:
        print(f"The capital of {state} is {capitals[state]}")

The capital of California is Sacramento
The capital of New York is Albany
The capital of Colorado is Denver
```

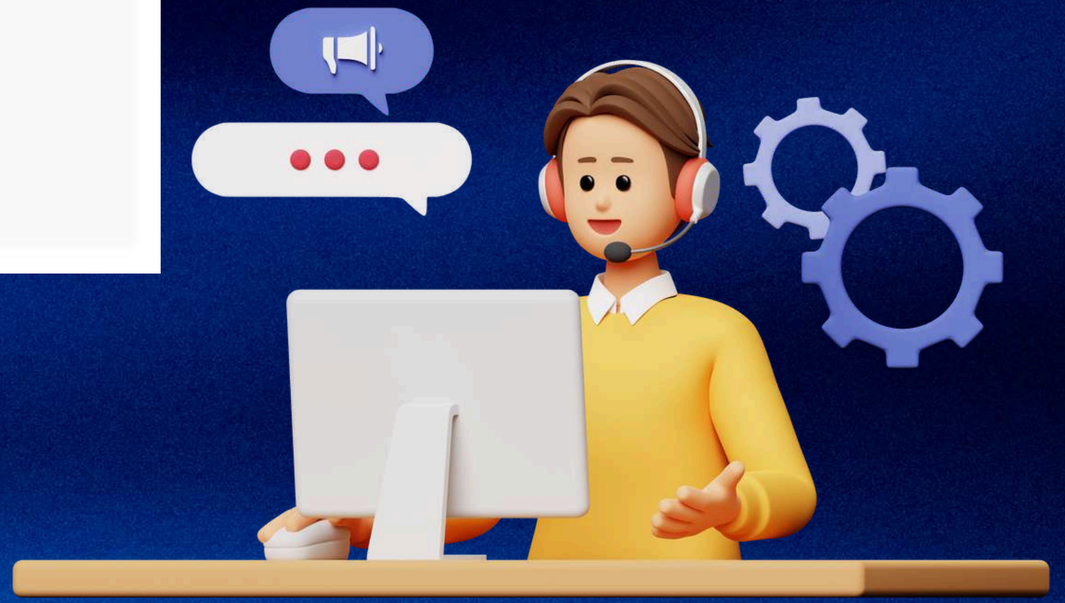
```
>>> capitals.items()
dict_items([('California', 'Sacramento'), ('New York', 'Albany'),
('Colorado', 'Denver')])
```

```
>>> type(capitals.items())
<class 'dict_items'>
```

DICTIONARY KEYS AND IMMUTABILITY

```
>>> capitals[50] = "Honolulu"
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany',
'Colorado': 'Denver', 50: 'Honolulu'}
```

```
>>> capitals[[1, 2, 3]] = "Bad"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```



Valid Dictionary Key Types

integers
floats
strings
booleans
tuples

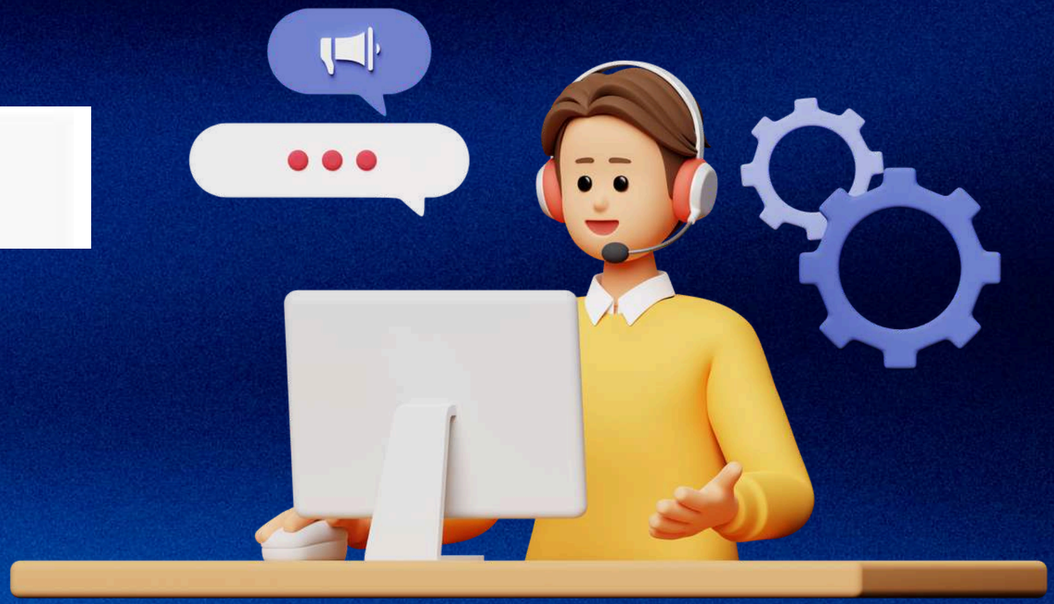
NESTED DICTIONARIES

THE VALUE OF EACH KEY IS A DICTIONARY:

```
>>> states["Texas"]
{'capital': 'Austin', 'flower': 'Bluebonnet'}
```

TO GET THE TEXAS STATE FLOWER, FIRST GET THE VALUE AT THE KEY "TEXAS", AND THEN THE VALUE AT THE KEY "FLOWER":

```
>>> states["Texas"]["flower"]
'Bluebonnet'
```



REVIEW EXERCISES

1. CREATE AN EMPTY DICTIONARY NAMED CAPTAINS.

2. USING THE SQUARE BRACKET NOTATION, ENTER THE FOLLOWING DATA INTO THE DICTIONARY, ONE ITEM AT A TIME:

```
'Enterprise': 'Picard'
'Voyager': 'Janeway'
'Defiant': 'Sisko'
```

3. WRITE TWO IF STATEMENTS THAT CHECK IF "ENTERPRISE" AND "DISCOVERY" EXIST AS KEYS IN THE DICTIONARY. SET THEIR VALUES TO "UNKNOWN" IF THE KEY DOES NOT EXIST.

4. WRITE A FOR LOOP TO DISPLAY THE SHIP AND CAPTAIN NAMES CONTAINED IN THE DICTIONARY. FOR EXAMPLE, THE OUTPUT SHOULD LOOK SOMETHING LIKE THIS:

```
The Enterprise is captained by Picard.
```

REVIEW EXERCISES

5. DELETE "DISCOVERY" FROM THE DICTIONARY.

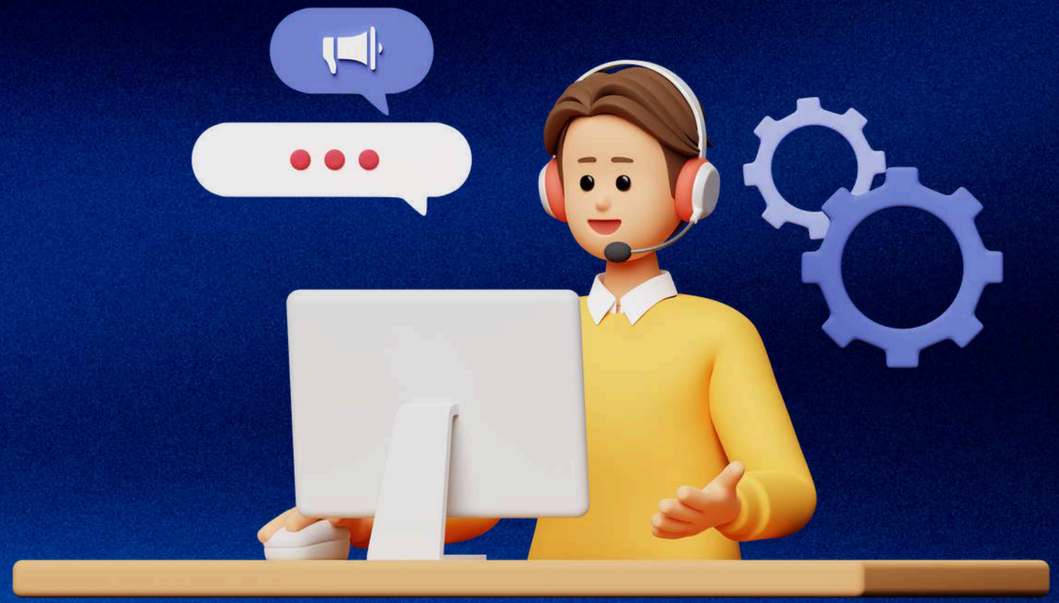
6. BONUS: MAKE THE SAME DICTIONARY BY USING DICT() AND PASSING IN THE INITIAL VALUES WHEN YOU FIRST CREATE THE DICTIONARY.

CHALLENGE: CAPITAL CITY LOOP

```
capitals_dict = {
    'Alabama': 'Montgomery',
    'Alaska': 'Juneau',
    'Arizona': 'Phoenix',
    'Arkansas': 'Little Rock',
    'California': 'Sacramento',
    'Colorado': 'Denver',
    'Connecticut': 'Hartford',
    'Delaware': 'Dover',
    'Florida': 'Tallahassee',
    'Georgia': 'Atlanta',
}
```

Note

Make sure the user is not punished for case sensitivity. In other words, a guess of `"Denver"` is the same as `"denver"`. Do the same for exiting—`"EXIT"` and `"Exit"` should work the same as `"exit"`.



HOW TO PICK A DATA STRUCTURE

USE A LIST WHEN:

- DATAHAS ANATURAL ORDER TO IT
- YOUWILL NEED TO UPDATE OR ALTER THE DATA DURING THE PROGRAM
- THEPRIMARY PURPOSE OF THE DATA STRUCTURE IS ITERATION

USE A TUPLE WHEN:

- DATA HAS ANATURAL ORDER TO IT
- YOUWILL NOTNEEDTOUPDATE OR ALTER THE DATA DURING THE PROGRAM
- THE PRIMARY PURPOSE OF THE DATA STRUCTURE IS ITERATION

USE A DICTIONARY WHEN:

- THEDATA IS UNORDERED, OR THE ORDER DOES NOT MATTER
- YOUWILL NEED TO UPDATE OR ALTER THE DATA DURING THE PROGRAM
- THEPRIMARY PURPOSE OF THE DATA STRUCTURE IS LOOKING UP VALUES

THANK YOU